Title: Introduction to Special Issue on Source Code Analysis and Manipulation.
Authors: Yoshiki Higo (Osaka University, Japan), Alexander Serebrenik (Eindhoven University of Technology, The Netherlands)

We are delighted to present this special issue dedicated to source code analysis and manipulation. Source code is the core of any software system: manually written or generated, textual or visual, it is a fully executable description of a software system. While research touching on source code is published in virtually every software engineering conference and journal, the IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM), an annual conference is explicitly dedicated to solidify and give shape to this research area.

The special issue has its roots in the 2019 edition of SCAM held in Cleveland, Ohao, USA. We encouraged the authors of papers presented at the conference to extend their work for this special issue by providing at least 30% of new material compared to their original works. We have also launched an open call for contributions on all aspects of source code.

The call for papers attracted 16 submissions covering a diverse range of relevant topics. Six of them are extended versions of SCAM 2019 papers. Each submitted article was carefully reviewed by at least three experts in the field. Eight excellent research papers have been accepted for publication in the special issue. The topics discussed in these papers cover a broad spectrum of manual and automatic code analysis and manipulation techniques: from refactoring to decompilation, from automatic fixing of static analysis warnings to web hybridisation, and from API usage to static single assignment programs.

In the article entitled “Java Decompiler Diversity and its Application to Meta-decompilation” Harrand et al. study the effectiveness of eight Java decompilers with respect to three quality indicators: syntactic correctness, syntactic distortion and semantic equivalence modulo inputs. The results show that no single modern decompiler is able to correctly handle the variety of bytecode structures coming from real-world programs. Even the highest ranking decompiler in this study produces syntactically correct output for 84% of classes of our dataset and semantically equivalent code output for 78% of classes. Furthermore the authors demonstrate that each decompiler correctly handles a different set of bytecode classes. Finally, Harrand et al. build a new decompiler that leverages the diversity of existing decompilers. To do so, the authors merge partial decompilation into a new one based on compilation errors. The new decompiler handles 37.6% of bytecode classes that were previously handled by no decompiler. Source code of the new bytecode decompiler has been made public.

Another paper included in this special issue is authored by Tiwari et al. entitled “A Large Scale Analysis of Android – Web Hybridization”. The authors analyze and categorize the parameters to web hybridization APIs for 7,500 randomly selected and the 196 most popular applications from the Google Playstore as well as 1000 malware samples. On this collection of applications the authors discovered thousands of flows of sensitive data from Android to JavaScript, the vast

majority of which could flow to potentially untrustworthy code. Additionally, the authors discovered a multitude of applications in which potentially untrusted JavaScript code may interfere with (trusted) Android objects, both in benign and malign applications. These results advance the general understanding of hybrid applications, as well as implications for potential program analyses.

The third paper included in this special issue is authored by Tahmooresi et al. and entitled "Studying the Relationship Between the Usage of APIs Discussed in the Crowd and Post-Release Defects". Tahmooresi et al. study whether using APIs which are challenging according to the discussions of the Stack Overflow is related to code quality defined in terms of post-release defects. The authors define the concept of API being challenging as the amount of discussion of this API in high-quality posts on Stack Overflow. Then, using this concept, Tahmooresi et al. propose a set of products and process metrics. The authors empirically study the statistical correlation between the metrics proposed and post-release defects as well as report on a case study on five open source projects including Spring, Elastic Search, Jenkins, K-8 Mail Android Client, and OwnCloud Android client. The results suggest that the metrics proposed have a positive correlation with post-release defects which is comparable to traditional process metrics, such as code churn and number of pre-release defects. The results suggest that software developers should consider allocating more resources to reviewing and improving external API usages as means of preventing further defects.

Masud et al. have focussed on a more precise construction of static single assignment programs using reaching definitions. The authors introduce an innovative φ-placement algorithm based on computing reaching definitions (RD), which generates a precise number of φ-functions, in the static single assignment form. The authors evaluate the algorithm theoretically by proving its correctness and ananlysing its computational complexity, as well as empirically by implementing it and a well-known DF-based algorithm in the Clang/LLVM compiler framework, and performed experiments on a number of benchmarks. The results show that the limiting assumption of the DF-based algorithm when compared with the more accurate results of our RD-based approach leads to generating up to 87% (69% on average) superfluous φ-functions on all benchmarks, and thus brings about a significant precision loss. Moreover, even though the approach proposed by Masud et al. computes more information to generate precise results, it is able to analyze up to 92.96% procedures (65.63% on average) of all benchmarks with execution time within 2x the execution time of the reference DF-based approach.

The fifth paper included in this special issue is authored by Marcilio et al. and entitled "SpongeBugs: Automatically Generating Fix Suggestions in Response to Static Code Analysis Warnings". In this paper, the authors investigate whether it is feasible to automatically generate fix suggestions for common warnings issued by static code analysis tools such as FindBugs and SonarQube, and to what extent developers are willing to accept such suggestions into the codebases they are maintaining. To this end, the authors implemented SpongeBugs, a Java program transformation technique that fixes 11 distinct rules checked by two well-known static

code analysis tools (SonarQube and SpotBugs). Fix suggestions are generated automatically based on templates, which are instantiated in a way that removes the source of the warnings; templates for some rules are even capable of producing multi-line patches. Based on the suggestions provided by SpongeBugs, the authors submitted to various Java open-source projects 38 pull requests, totalling 946 fixes. The project maintainers accepted 87% of the fix suggestions (97% of them without any modifications). The authors further evaluated the applicability of their technique to software written by students and to a curated collection of bugs. All results indicate that the approach to generating fix suggestions is feasible, flexible, and can help increase the applicability of static code analysis tools.

The sixth paper included in this special issue is authored by Peruma et al. and entitled "Contextualizing Rename Decisions using Refactorings, Commit Messages, and Data Types". In this paper, the authors extend their prior work, focussing on rename refactorings. In the prior work, the authors contextualize rename changes by examining commit messages and other refactorings. In this extension, the authors further consider data type changes which co-occur with the renames, with a goal of understanding how data type changes influence the structure and semantics of renames. In the long term, the outcomes of this study will be used to support research into: 1) recommending when a rename should be applied, 2) recommending how to rename an identifier, and 3) developing a model that describes how developers mentally synergize names using domain and project knowledge. The authors provide insights into how their data can support rename recommendation and analysis in the future, and reflect on the significant challenges, highlighted by their study, for future research in recommending renames.

The seventh paper included in this special issue is authored by Alomar et al. and entitled "Toward the Automatic Classification of Self-Affirmed Refactoring". The act of intentionally documenting a refactoring activity is known as Self-Affirmed Refactoring (SAR). This paper aims to automate the detection and classification of refactoring documentation in commit messages. Specifically, the authors combine the N-Gram TF-IDF feature selection with binary and multiclass classifiers to build a new model to automate the classification of refactorings based on their quality improvement categories. The authors apply the model to 2,867 commit messages extracted from well-engineered open-source Java projects. Their findings show that (1) the model is able to accurately classify SAR commits, outperforming the pattern-based and random classifier approaches, and allowing the discovery of 40 more relevant SAR patterns, and (2) the model reaches an F1-measure of up to 90% even with a relatively small training dataset.

The last paper included in this special issue is authored by Alanazi et al. and entitled "Facilitating Program Comprehension with Call Graph Multilevel Hierarchical Abstractions". This paper presents a coarse-grained technique for creating multi-level hierarchical representations of call graphs. Specifically, it proposes a technique for visualizing call graphs at different levels of granularity and for different software units, such as packages, classes, and functions, by hierarchically clustering execution paths. The goal of this paper is to enable software developers to understand software systems from high to low levels of abstraction and to focus on specific

parts of the system. The authors conducted a user study in which 18 software engineers from multiple companies were asked to perform several tasks using the developed system and then answer a questionnaire to validate the approach and tool support. The results demonstrate that the approach proposed is capable of automatically constructing multi-level abstractions of call graphs and hierarchically clustering them into meaningful abstractions.